Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)

Физтех-школа радиотехники и компьютерных технологий
Кафедра системного программирования ИСП РАН
Лаборатория (laboratory name)

Выпускная квалификационная работа бакалавра

Исследование и разработка методов машинного обучения

Автор:
Студент 082 группы
Иванов Иван Иванович

Научный руководитель:
научная степень
Денисов Денис Денисович

Научный консультант:
научная степень
Сергеев Сергей Сергеевич

Москва 2025

Аннотация

Исследование и разработка методов машинного обучения
Иванов Иван Иванович

Краткое описание задачи и основных результатов, мотивирующее прочитать весь текст.

Abstract

Research and development of machine learning methods

Содержание

1 Введение

Типовая система языка программирования служит фундаментом для формального определения его спецификации, обеспечивает статический анализ программ, поддержку оптимизаций кода и способствует улучшению его структурированности и читаемости. Высокоуровневые языки программирования(Java, C#, Kotlin) предоставляют разработчику мощные абстракции и безопасность кода посредством автоматического управления памятью и объектно-ориентированных парадигм. Многие из этих языков сохранили в себе исторически сложившееся разделение типов на "примитивные"типы и “объектные” или “ссылочные”, призванное потенциально улучшить производительность в некоторых случаях. Тем не менее, в то же время оно приводит к несогласованности в системе типов и усложняет достижение полностью единообразной и интуитивной модели программирования. Примитивные типы часто лишены полезных свойств объектов, таких как наследование, полиморфизм, вызов методов без явного упаковывания/распаковывания и использование в качестве аргумента для шаблонных типов.

В данной работе будут теоретически рассмотрены последствия устранения явной разницы между примитивными и объектными типами из высокоуровневого управляемого языка программирования и описаны этапы и результаты реализации.

1.1 Типовая система

Типовая система — это формальная синтаксическая и семантическая структура, сопоставляющая типы и программные конструкции (выражения, переменные, функции) для обеспечения чётко определённого вычислительного поведения. Её основные теоретические и практические назначения:

1) **Корректность поведения программы**

   С введением типовой системы становится возможной статическая верификация соблюдения ограничений типов, не допускающая к выполнению неверно типизированные программы. Данный механизм гарантирует:

   - *Безопасность выполнения* — предотвращение неопределённого поведения за счёт блокировки запрещённых операций (например, некорректные обращения к памяти в управляемых средах типа JVM/CLR)
   - *Теоретическую обоснованность* — соответствие двум фундаментальным принципам типобезопасности по Райту-Феллейзену:
     1. **Прогресс**: корректно типизированная программа не может застрять в промежуточном состоянии
     2. **Сохранение**: типы остаются согласованными на всех этапах вычисления

2) **Уровень абстракции кода**

Типовая система обеспечивает инкапсуляцию доменных инвариантов с помощью абстракций типов (абстрактные типы данных и интерфейсы) и позволяет формально специфицировать границы модулей.

3) **Возможность оптимизаций**

Наличие статической типизации позволяет применять оптимизации на этапе компиляции (например, специализация обобщённых типов и статическое разрешение методов) и минимизировать время исполнения в управляемых средах за счёт уменьшения количества динамических проверок типов.

1.2 Статическая и динамическая типизации

1. **Статическая типизация** (Java, C#, Kotlin):

   - Валидность типа доказывается на этапе компиляции с помощью формальных суждений о принадлежности выражения к типу в данном контексте ($\Gamma \vdash e : \tau$)
   - Гарантирует типобезопасность по Райту-Феллейзену
   - Критично для управляемых языков программирования из-за возможности проверок безопасности памяти (например, верификатор байт-кода JVM) и эффективности Just-In-Time компиляции за счёт встраивания методов, ориентированных на конкретный тип

2. **Динамическая типизация** (Python, JavaScript):

   - Типы функций и выражений определяются во время исполнения
   - Нет формальной гарантии типобезопасности, но есть возможность менять поведение объектов, модулей и классов во время исполнения

Современные управляемые языки высокого уровня (Java, C#) преимущественно основаны на статической типизации, расширенной возможностями динамического анализа во время выполнения. Данный компромиссный подход позволяет сочетать строгость формальной верификации с практической гибкостью разработки, но противоречие между статическими гарантиями и динамической адаптируемостью продолжает оставаться предметом активных исследований.

Исторически и архитектурно, многие языки разделяют типы на две категории: **Примитивные типы** (далее – примитивные типы или примитивы) Представляют собой базовые значения (например, целые числа, числа с плавающей точкой, булевы значения, символы). **Ссылочные типы** (далее - ссылочные типы, объектные типы)Представляют собой экземпляры классов, которые хранятся в куче (heap) и управляются сборщиком мусора. Они наследуются от базового класса и обладают методами, полями и другими объектно-ориентированными свойствами.

1.3 Примитивные типы

Примитивные типы являются наиболее фундаментальным типом данных в языках программирования. В отличие от объектных типов, они:

- **Не ссылают на память**, а непосредственно являются значениями. У них отсутствуют:
  - заголовок объекта (object header)
  - таблица виртуальных методов (vtable)
  - ссылочная идентичность (identity)
- Имеют **фиксированную семантику**, определяющуюся стандартами языка (например, Java JLS §4.2, C# ECMA-334 §8.3)
- **Не могут быть подтипами** или наследоваться и являются атомарными (базовыми) элементами в иерархии типов

1.3.1 Роль в конструировании типов

В теории типов примитивы являются простейшими случаями конструирования типа и не разлагаются на более простые типы. Конструирование сложных типов происходит на основе:

- **Произведения** (кортежи, записи) – (Int, Bool)
- **Суммы** (типы-суммы, перечисления) – `data Maybe a = Nothing | Just a`
- **Функции** – Int → Bool
- **Рекурсивные типы** – `data List a = Nil | Cons a (List a)`

1.3.2 Основные характеристики

- **Представление в памяти**

  Аллокация на стеке* (default for local variables in methods)

- **Производительность на уровне процессора**

  1. Арифметические операции с примитивами компилируются непосредственно в нативные инструкции:

```
    ; x86 assembly for 'a + b'
    mov eax, [a]
    add eax, [b]

```

  2. **Локализация в кэше**

     Линейная модель доступа к памяти (критично для больших вычислений)

1.3.3 Примеры в высокоуровневых языках – NEED TO BE FIXED

Таблица 1: Примитивные типы в различных языках программирования

Тип	**Java**	**C#**
Целые числа	`int`, `long`	`int`, `long`
Числа с плавающей	`float`, `double`	`float`, `double`
Булевы значения	`boolean`	`bool`
Символ	`char`	`char`

1.4 Роль примитивных типов в управляемых средах исполнения

Управляемая среда исполнения (Managed Execution Environment) — это программный компонент или слой, который обеспечивает выполнение программного кода в контролируемой, безопасной и абстрагированной от низкоуровневых деталей среде. Она действует как посредник между скомпилированным (или интерпретируемым) кодом приложения и базовой операционной системой или аппаратным обеспечением.

1.4.1 Основные функции и компоненты среды исполнения

Виртуальная Машина (ВМ) / Интерпретатор: Ядро среды исполнения, которое выполняет промежуточный код (байт-код или Common Intermediate Language – CIL). Примеры включают Java Virtual Machine (JVM) и Common Language Runtime (CLR) для .NET. ВМ изолирует исполняемый код от конкретной аппаратной архитектуры и операционной системы.

JIT-компиляция (Just-In-Time Compilation): Для повышения производительности многие ВМ используют JIT-компиляторы. Они динамически переводят промежуточный код в машинный код непосредственно перед его выполнением. JIT-компиляторы могут применять оптимизации на основе профилирования времени выполнения, что часто позволяет достичь производительности, сопоставимой с нативно скомпилированным кодом.

Загрузчики Классов/Сборок: Эти компоненты отвечают за динамическую загрузку программных модулей (классов, библиотек, сборок) по мере их необходимости во время выполнения программы. Они также управляют разрешением зависимостей и проверкой целостности загружаемых компонентов.

Система Управления Потоками: Среда исполнения предоставляет API и механизмы для создания, управления и синхронизации потоков выполнения, обеспечивая эффективную поддержку параллельных вычислений.

Система Безопасности: ВУЯП часто включают встроенные модели безопасности (например, песочницы), которые контролируют доступ исполняемого кода к системным ресурсам, таким как файловая система, сеть или другие процессы, что особенно важно для кода, загружаемого из недоверенных источников.

Встроенные Сервисы: К ним относятся механизмы обработки исключений, рефлексия (возможность интроспекции и модификации структуры кода во время выполнения), сериализация, а также поддержка взаимодействия с нативным кодом (Native Interface).

1.4.2 Примитивные типы в управляемых средах исполнения

Примитивные типы данных в контексте управляемых сред исполнения (таких как .NET CLR и Java VM) обладают характеристиками и принципами работы, отличающимися от неуправляемых сред, например, реализованных на языках C/C++.

Абстракция и безопасность

Управляемые среды исполнения предоставляют уровень абстракции, скрывающий детали низкоуровневого представления примитивных типов данных в памяти. Это обеспечивает платформонезависимость и переносимость программного обеспечения. Кроме того, такие среды реализуют строгую проверку типов как на этапе компиляции, так и во время исполнения, что служит профилактикой ошибочного использования типов данных и возникновения уязвимостей, связанных с выделением памяти и выходом за границы допустимых областей.

Гарантированная инициализация

В отличие от неуправляемых сред, где переменные могут содержать неинициализированные значения, управляемые среды гарантируют автоматическую инициализацию переменных примитивных типов значениями по умолчанию. Это исключает возможность непредсказуемого поведения, связанного с использованием мусорных значений памяти.

Сборка мусора

В управляемых средах примитивные типы данных, как правило, не подлежат сборке мусора напрямую. Обычно они размещаются в стеке либо, если выступают в роли компонентов объектов, внутри кучи. Освобождение памяти, занятой объектами (соответственно, и их полями-примитивами), происходит посредством встроенного механизма сборки мусора, что снимает с разработчика необходимость вручную управлять памятью.

Стандартное поведение

Управляемые среды исполнения гарантируют предсказуемое и стандартизированное поведение всех операций с примитивными типами данных, в том числе арифметических. Данная особенность повышает надежность и упрощает переносимость программ между различными аппаратными и программными платформами.

Механизм упаковки и распаковки (boxing/unboxing)

Управляемые среды поддерживают механизм упаковки (*boxing*) и распаковки (*unboxing*), позволяющий преобразовывать значения примитивных типов в объекты и обратно. Это особенно актуально при взаимодействии с универсальными коллекциями и другими структурами, требующими объектного представления данных. Например, в среде C# значение типа `int` может быть преобразовано в объект (*boxing*) и обратно (*unboxing*).

Примеры реализации

- **.NET (например, на C#)**: Примитивные типы (такие как `System.Int32`) реализованы в виде структур и обеспечивают предсказуемое поведение за счёт встроенных механизмов среды.
- **Java**: Примитивные типы (`int`, `boolean`, `char`) имеют фиксированные размеры и поведение. Для поддержки упаковки и распаковки используются соответствующие классы-обёртки (`Integer`, `Boolean`, `Character`).

1.5 Несогласованность примитивных типов в управляемых языках программирования

В большинстве управляемых языков программирования (таких как Java, C#, Python) существует проблема несогласованности обработки примитивных типов данных и объектных типов в рамках единой системы типов. Эта асимметрия оказывает существенное влияние на архитектуру языка, его производительность и способность к применению принципов объектно-ориентированного программирования.

Отсутствие единообразия в системе типов

Одной из ключевых проблем является отсутствие унифицированного представления примитивных и объектных типов в системе типов управляемого языка. Это бинарное разделение приводит к концептуальной и синтаксической асимметрии. Например, примитивные типы не могут быть `null` (без дополнительных оберток), не поддерживают полиморфизм и не могут использоваться там, где требуется экземпляр класса `Object` (например, в универсальных коллекциях) без явного или неявного преобразования.

Ограничения в Объектно-Ориентированных Возможностях

Несогласованность примитивных типов создает существенные ограничения для применения объектно-ориентированных парадигм: инкапсуляция, наследование и полиморфизм.

- **Отсутствие наследования и полиморфизма:** Примитивные типы не могут наследоваться, и к ним не применимы механизмы полиморфизма через интерфейсы или абстрактные классы. Это означает, что функции, ожидающие экземпляр базового класса или интерфейса, не могут напрямую работать с примитивными значениями.

- **Проблемы с коллекциями и обобщениями (Generics):** Для включения примитивных значений в объектные коллекции (например, `ArrayList<Object>` в Java или `List<object>` в C#) требуется механизм автоматической или явной упаковки (boxing) в соответствующие объектные обертки (например, `Integer` для `int`, `Double` для `double`). Это не только нарушает прозрачность и чистоту кода, но и может приводить к неожиданным побочным эффектам или ошибкам типизации, если разработчик не учитывает процесс упаковки/распаковки.

- **Нарушение унифицированного доступа:** Объектно-ориентированный дизайн стремится к унифицированному доступу к данным и поведению через методы. Примитивные типы не обладают методами, так что разработчик должен использовать процедурные подходы или статические вспомогательные классы для выполнения операций над ними.

Производительность и накладные расходы

Примитивные типы, как было описано выше, позволяют достичь высокой производительности из-за их прямой аллокации в стеке или регистрах. Тем не менее, их несогласованность с объектной системой может приводить к значительным накладным расходам.

- **Операции упаковки и распаковки (Boxing/Unboxing):** Эти операции, интегрирующие примитивы в объектную иерархию, влекут за собой:
  - **Аллокацию памяти в куче:** Для каждого упакованного примитивного значения создается новый объект в куче, что увеличивает потребление памяти.
  - **Дополнительные циклы процессора:** Создание и инициализация объектов-оберток, а также последующее их удаление сборщиком мусора, требуют процессорного времени.
  - **Увеличение нагрузки на сборщик мусора:** Большое количество короткоживущих объектов-оберток создает дополнительную работу для сборщика мусора, потенциально приводя к задержкам в работе приложения.
- **Избыточные преобразования типов:** В сложных системах, где примитивы часто передаются между функциями, ожидающими объектные типы, и наоборот, могут возникать множественные операции упаковки и распаковки, что негативно сказывается на общей производительности системы.

Влияние на проектирование языка и его особенности

Асимметрия между примитивными и объектными типами оказывает глубокое влияние на архитектуру и проектирование самих управляемых языков, усложняя их реализацию и использование.

- **Сложность дизайна и реализации компилятора/рантайма:** Разработчикам языков приходится вводить специальные правила и исключения для обработки при-

митивных типов, которые не соответствуют общей объектной модели. Это увеличивает сложность компилятора и среды исполнения, требуя специализированных путей кода для различных типов.

- **Ограничения в расширяемости:** Механизмы расширения языка, такие как операторная перегрузка или метапрограммирование, могут быть ограничены или усложнены из-за необходимости учитывать различия между примитивами и объектами.
- **Влияние на ключевые функции языка:**
  - **Рефлексия:** Механизмы рефлексии должны предоставлять отдельные API для примитивных и объектных типов или вводить специальные обертки для работы с примитивами.
  - **Сериализация:** Унифицированная сериализация данных становится более сложной, поскольку необходимо обрабатывать как объекты, так и примитивные значения.
  - **Нулевые типы (Nullable Types):** Введение безопасных nullable-типов (например, `int?` в C#) часто требует дополнительной работы для примитивов, в то время как объектные типы по умолчанию могут быть `null`.
- **Дополнительная нагрузка на разработчика:** Разработчики, использующие управляемые языки, вынуждены постоянно учитывать различия между примитивами и объектами и помнить, когда и где следует использовать примитив, а когда — его объектную обертку, добавляет сложности в процесс разработки.

2 Постановка задачи

2.1 Контекст

В данной работе речь пойдет о высокоуровневом языке программирования с управляемой средой исполнения, поддерживающей два языка программирования — статически типизированный язык, похожий в большей степени на TypeScript, и динамически типизированный, схожий с JavaScript.

Язык поддерживает императивные, объектно-ориентированные, функциональные и шаблонные паттерны программирования и комбинирует разные семантические аспекты TypeScript, Java и Kotlin. На данный момент язык находится в активной стадии разработки.

Примитивные типы в разрабатываемом языке обладают основными характеристиками, описанными ранее — они:

- не участвуют в подтипировании (в том числе с типом `Object`)
- не могут быть компонентами юнион-типов
- не могут приниматься в generic-типах в качестве аргумента
- не имеют методов и других свойств объектов

Для обеспечения указанной функциональности язык предоставляет “Большебуквенные” аналоги примитивных типов, представляющие собой объектную обертку с соответствующими методами и правилами наследования (например, `Int` — объектный аналог примитивного типа `int`). В случаях необходимости между примитивным типом и его аналогом происходит скрытая конверсия, что усложняет семантический анализ программ. Более того, синтаксически допускаются конструкции вида `int | undefined` или `Set<int>`, которые интерпретируются компилятором как `Int | undefined` и `Set<Int>`. Несоответствие свойств `int` и `Int` приводит к непредвиденному поведению в краевых случаях.

Далее в тексте `int` и `Int` будут использоваться в качестве основного примера.

Цель работы: улучшение семантического анализа программ посредством устранения концепции примитивных типов как отдельной категории и унификации системы типов.

2.2 Задачи исследования

Для достижения поставленной цели необходимо решить следующие задачи:

1. Провести детальный анализ текущего состояния семантического анализа с целью идентификации мест, где наличие примитивных типов приводит к:
   - усложнению логики анализа
   - неоднозначности интерпретации

- необходимости специальных правил обработки (скрытые конверсии, union-типы, generics)

2. Разработать формальную спецификацию унифицированной системы типов, которая:

- исключает концепцию примитивных типов
- интегрирует их функциональность в объектную модель
- сохраняет производительность операций (арифметические, логические)
- обеспечивает корректность семантики

3. Спроектировать и реализовать модификации в подсистеме семантического анализа, обеспечивающие:

- проверку совместимости типов
- разрешение перегрузок
- вывод типов
- работу с метаданными

4. Разработать набор тестовых сценариев, охватывающих:

- generic-типы с числовыми аргументами
- union-типы
- операции с числовыми значениями
- проблемные краевые случаи

5. Оценить влияние решения на:

- сложность и точность семантического анализа
- удобство разработки
- потенциальную производительность

с использованием формальных метрик:

- уменьшение количества специальных правил
- сокращение ветвлений в коде анализатора
- результаты тестовых сценариев

2.3 Требования к решению

2.3.1 Функциональные требования

1. Унификация системы типов: Разработанное решение должно обеспечить:

- полную интеграцию всех типов (числовых, логических) в общую объектную иерархию
- участие базовых типов в механизмах подтипирования
- устранение разделения на примитивные и объектные типы

2. **Поддержка продвинутых конструкций:** Все типы должны корректно использоваться в:
   - **Union-типах:** конструкции вида `number | undefined` или `boolean | null` должны быть валидными
   - **Дженериках:** базовые типы как аргументы (`Set<number>`, `Map<string, boolean>`) без скрытых преобразований
   - **Наследовании/Подтипировании:** соблюдение общих правил для базовых типов
3. **Устранение скрытых конверсий:** Необходимо исключить:
   - автоматические механизмы упаковки (boxing)
   - автоматическую распаковку (unboxing) значений
   - неявные преобразования между разными представлениями типов
4. **Семантическая корректность:** Анализатор должен обеспечивать:
   - точное выявление типовых ошибок
   - однозначное разрешение типов в выражениях
   - предсказуемое поведение программ
5. **Сохранение синтаксиса:** Требуется:
   - сохранить существующий синтаксис (`int`, `number`, `boolean`)
   - обеспечить соответствие внутренней интерпретации унифицированной модели

2.3.2 Нефункциональные требования

1. **Производительность:**
   - отсутствие существенного замедления семантического анализа
   - оптимизация алгоритмов проверки типов
2. **Эффективность кода:**
   - отсутствие заметного снижения производительности исполняемого кода
   - сравнимая эффективность с оригинальной реализацией
3. **Качество кода анализатора:**

- повышение читаемости и поддерживаемости
- модульная структура
- упрощение логики обработки типов

4. **Удобство разработки:**
   - интуитивно понятная модель типов
   - снижение когнитивной нагрузки
   - минимизация неочевидных ошибок

2.4 Ожидаемые результаты

В результате выполнения дипломной работы ожидается получить:

- **Спецификацию унифицированной системы типов** для высокоуровневого управляемого языка, включающую:
  - формальное описание типовой системы
  - правила подтипирования
  - алгоритмы проверки типов
- **Модифицированную подсистему семантического анализа** с:
  - поддержкой унифицированных типов
  - устранением специальных случаев для примитивов
  - улучшенной архитектурой
- **Комплекс тестовых примеров**, покрывающих:
  - базовые операции с типами
  - union-типы и дженерики
  - краевые случаи
  - производительность анализа
- **Оценочные метрики** по:
  - сложности семантического анализа
  - эффективности работы компилятора
  - удобству использования языка

3 Обзор существующих решений

3.1 Интеграция примитивных типов в высокоуровневую систему типов и их влияние на семантический анализ

3.1.1 Стратегия: Явное разделение с автоматическим преобразованием (Autoboxing/Autounboxing)

- **Суть:** Примитивные типы (‘int‘, ‘float‘, ‘boolean‘) и объектные типы (‘Integer‘, ‘Float‘, ‘Boolean‘) существуют параллельно и явно различимы в системе типов языка. Компилятор автоматически вставляет преобразования (боксинг - примитив -> объектная обертка, анбоксинг - объектная обертка -> примитив) там, где контекст требует типа другого вида (например, передача ‘int‘ в метод, ожидающий ‘Object‘, или использование ‘Integer‘ в арифметической операции).
  - Боксинг: примитив → объектная обёртка
  - Анбоксинг: объектная обёртка → примитив
- **Влияние на систему типов:**
  - Явный дуализм типов
  - Примитивы не являются частью объектной иерархии
  - Необходимость учёта обоих "миров"в правилах подтипирования
- **Влияние на семантический анализ:** Значительно усложняет анализ
  - Отслеживание контекстов, требующих преобразований
  - Разрешение перегрузок методов с разными параметрами
  - Обработка потенциальных `NullPointerException`
  - Учёт различий в семантике операторов (напр., `==`)
  - Отдельные ветви кода для проверки типов
- **Преимущества:**
  - Производительность примитивов
  - Полиморфность объектных обёрток
  - Понятность для разработчиков
- **Недостатки:**
  - Сложная система типов
  - Риск `NullPointerException`

  - Накладные расходы на преобразования
  - Концептуальный разрыв

- **Примеры:** Java (классический пример), ранние версии C#

Таблица 2: Сравнение поведения операторов

Операция	**Семантика**
== для примитивов	Сравнение значений
== для объектов	Сравнение ссылок
Арифметические операции	Автоматический анбоксинг

```
Integer x = 10;   // Автобоксинг
int y = x;        // Автоанбоксинг
if (y == x) {     // Сравнение с автоанбоксингом
    System.out.println("Equal");
}
```

Листинг 1: Пример в Java